THE AUSTRALIAN NATIONAL UNIVERSITY

Building Fast, Reliable and Adaptive Software for Computational Science

Alistair Rendell, Joseph Antony, Warren Armstrong,
Pete Janes and Rui Yang
Dept. Of Computer Science
Australian National University

As we have been hearing...

After a decade of incremental change we now have:

- Multicore on the desktop
- Easily (?) programmable GPUs
  - NVIDIA GTX 8800 GPU with CUDA programming interface
- Special purpose processors
  - Cell Broadband Engine/Playstation 3, cheap FPGAs
- Machines like the Sun ROCKS system with transactional memory

Given this hardware maze

- How do we construct computational science applications that are:
  - Fast
  - Reliable
  - Adaptive
- Some overlap with goals of SciDAC Performance Engineering Research Institute (PERI)

ANU

Fast, Reliable and Adaptive

- RELIABLE: Numerically reliable on petaflop systems
  - Use of interval arithmetic to place rigorous bounds on complex numerical calculations
- FAST: Performance models to predict the effect of runtime modifications
  - Models to capture cache usage and memory placement effects
- ADAPTIVE: Able to respond to a changing runtime environment
  - Use of dynamic code modification to alter the behavior of a running application and machine learning to predict those changes

ANU

RELIABLE
Interval Arithmetic to Track Numerical Errors

Objective: To track rigorously the truncation and rounding errors that occur in large scale electronic structure computations?

Computational Errors

- Modeling approximations
  - Classical instead of quantum mechanics
- Truncation errors
  - Due to algorithmic approximations, e.g. series truncation

$$e \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!}$$

- Rounding errors
  - ~10^{-8} single precision, ~10^{-16} double precision increases with operations
  - Subtraction particularly problematic

With petaflop computers performing 10^{15} operations per second how will we know if anything we compute is correct?

Precision ≠ Accuracy

- **Rump's example**

$$f = (333.75 - a^2)b^6 + a^2(11a^2b^2 - 121b^4 - 2) + 5.5b^8 + a/(2b)$$

$$a = 77617$$

$$b = 33096$$

```
 32 bit: f = 1.172604

 64 bit: f = 1.1726039400531786

128 bit: f = 1.172603940053178631858834904520183 8

correct: f = -.827396059946821368141165095479816...
```

Intervals

- Not new, concept been around since early floating point
  - Sun Fortran compiler provides an interval data type
- Guaranteed error bounds computed with results
- Represented as two numbers

$$[a,b] = \{x | a \le x \le b\}$$
$$\equiv \hat{x} \pm \varepsilon \text{ where } \quad a = (\hat{x} - \varepsilon), b = (\hat{x} + \varepsilon)$$

$$X \circ Y \equiv \{x \circ y | x \subset X; y \subset Y\}$$
$$\circ \in \{+, -, \times, /\}$$

- Interval arithmetic: given two intervals

$$[\underline{x}, \overline{x}] + [\underline{y}, \overline{y}] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$$
$$[\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}] = [\min(\underline{x} \times \underline{y}, \underline{x} \times \overline{y}, \overline{x} \times \underline{y}, \overline{x} \times \overline{y}),$$
$$\max(\underline{x} \times \underline{y}, \underline{x} \times \overline{y}, \overline{x} \times \underline{y}, \overline{x} \times \overline{y})]$$

Before you ask...

- **Rump's Example Using Intervals**

$$f = (333.75 - a^2)b^6 + a^2(11a^2b^2 - 121b^4 - 2) + 5.5b^8 + a/(2b)$$

$$a = 77617$$

$$b = 33096$$

```
 32 Bit: [-1.901..E+30,2.535..E+30] width 1E60
 64 Bit: [-4.722..E+21,5.902..E+21] width 1E42
128 Bit: [-5.118..E+03,4.097..E+03] width 1E06

correct: f = -.827396059946821368141165095479816...
```

- **Not great, but at least we know we have a problem!**

Intervals in Electronic Structure Methods

- At the core of most algorithms for computing the sorts of integrals used in electronic structure codes is evaluation of the incomplete gamma function:

$$F_m(T) = \int_0^1 t^{2m} e^{-Tt^2} dt$$

- The value of this is given by an infinite series
- Programs use either finite series evaluation or an asymptotic approximation
  - Switch depends on value of T for a given value of m
- To speed evaluation interpolation is used
  - Chebyshev and Taylor interpolation over uniformly discretized domain

$F_m(T)$ Evaluation Errors

- Fundamental approximations made before any computation has begun
  - Finite series v asymptotic approximation
  - Type of interpolation

These give rise to Truncation errors

- How we perform the numerical operations
  - Order that we manipulate the data
  - Simple summation v use of compensated summation in series evaluation

These give rise to Rounding Errors

- Using interval arithmetic we can bound both errors

$F_m(T)$: Average Relative Error (x1e15)

- **Defined as**
  - **Interval width divided by absolute value of midpoint**

Evaluation Scheme		m = 0	m = 4	m = 8	m = 12
ChebyA	Alg1	83	4300	22000	67000
	Alg2	32	1600	7900	23000
ChebyB	Alg1	14	17	19	21
	Alg2	6	7	8	9
ChebyC	Alg1	15	18	20	21
	Alg2	7	9	9	10
TaylorA	Alg1	14	17	18	20
	Alg2	5	6	7	8
TaylorB	Alg1	14	17	19	20
	Alg2	6	7	8	9

ANU

Current Work

- Different approaches to accurate floating point summation
  - Evaluation of the electrostatic energy for a group of point charges
  - Comparison of pairwise summation with fast multipole methods
- Extend $F_m(T)$ to full integral evaluation and then to a complete Hartree-Fock (HF) code
  - We have looked at alternative approaches to integral evaluation, full HF is under development
- Other uses for intervals
  - There are novel interval algorithms for global optimization, which we are exploring in the context of solving the HF equations and locating the global minimum on a potential energy surface

FAST
Performance Models for Cache Behavior

Objective: To develop a simple model for cache behavior that is accurate enough to provide predictive information for use in, for example algorithm selection or cache blocking?

ANU

Cache Performance Models

- Essentially two approaches
  - Analytical or Simulation Based
- Analytical
  - Parameterize system to give empirical performance estimates
  - Fail to capture dynamic nature of code execution
- Simulation based
  - Predict performance based on sequence of executable instructions
  - Instructions can be execution or trace driven
  - 100-1000 times slower than execution on native hardware

Linear Performance Model (LPM)

$$time = \alpha \times I\_count + \beta \times Cache\_Miss$$

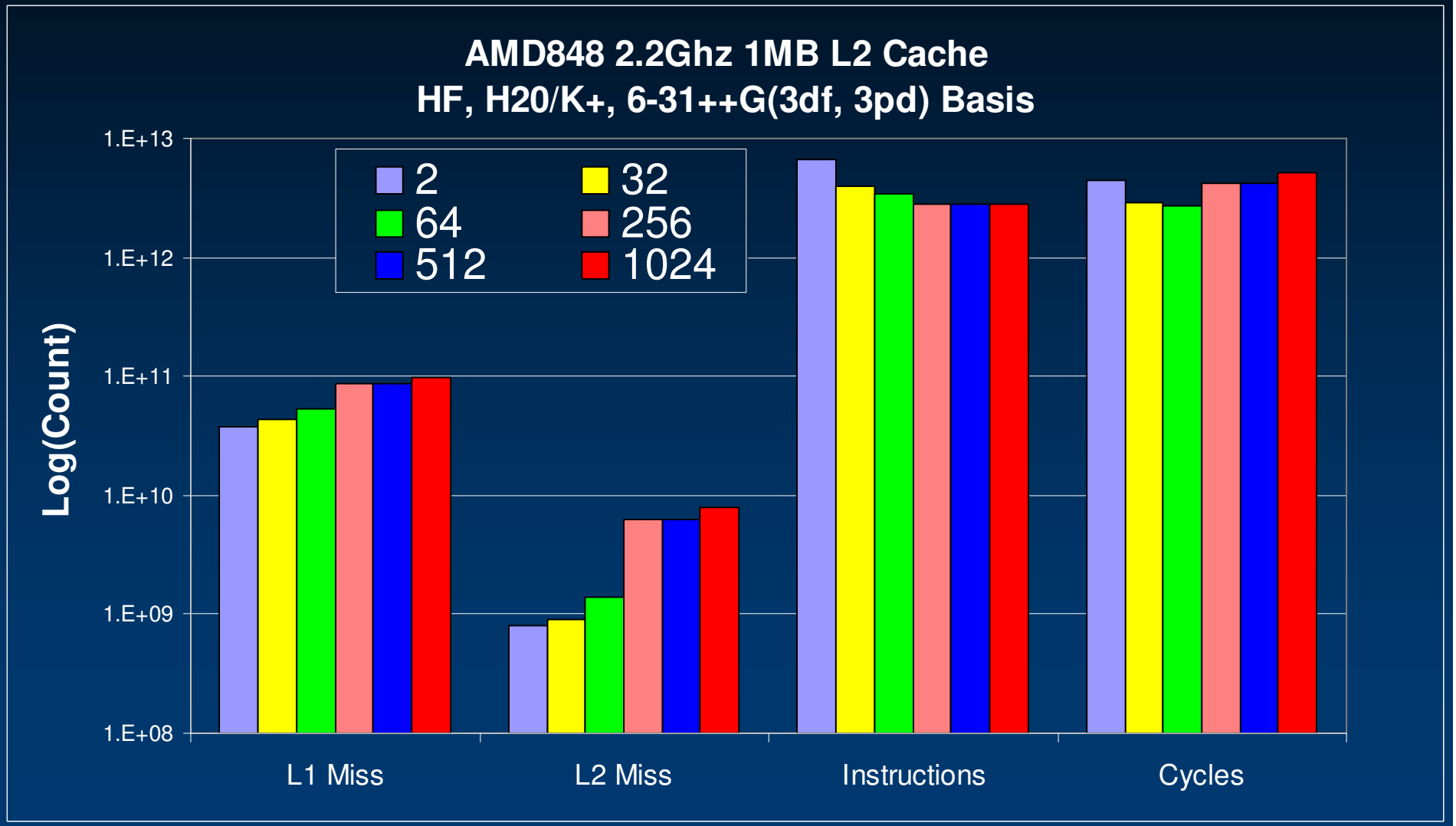
`I_count`	Instruction Count	Measured
`Cache_Miss`	Level 1/2 Cache Misses	Measured
α and β	Penalties	Fitted

- LPM ignores intricacies of program execution
  - average the details into fitting parameters α and β
  - expect to work best for "similar" calculations
- How to obtain α and β
  - run different calculations with different input data sets
  - run one calculation and alter cache usage to obtain different counts
  - perform least squares fit from data

Test Case: Integral Evaluation

- In nearly every electronic structure calculation
  - associated with electron/electron repulsion, electron/nuclei attraction (what we compute using $F_m(T)$)
  - accounts for 80-90% of time in typical HF computation
- The PRISM integral evaluation algorithm computes integrals in batches
  - batches contain integrals of a similar type
- For large systems batches can become very large
  - good for vector machine, poor for scalar/cache machine
- To enhance performance on scalar machines a maximum batch size is imposed
  - usually determined based on cache size in an ad-hoc fashion

Counts as Function of Cache Blocking



LPM Parameters by CPU

		Instructions ($1/\alpha$) per cycle			Cache Misses (γ) (Cycles)		
CPU	GHz	Lim	Avg	STD	Exp	Avg	STD
Athlon64	2.2	3	1.7	0.08	238	634.2	82.8
Opteron	2.2	3	1.7	0.03	300	385.9	55.9
EMT64T	3.0	3	1.0	0.08	442	115.5	24.1
Pentium 4	3.0	3	0.9	0.05	430	214.1	24.2
Pentium M	1.4	3	1.4	0.04	204	72.0	13.4
Apple G5	1.8	3	1.4	0.06	692	442.4	20.8

- **Exp:**
  - **Experimental data measured using lmbench**
- **Data averaged over 4 different calculations**

Accuracy of LPM (% Relative Error)

	Molecular System				
CPU	1	2	3	4	5
Opteron	3.2	1.0	2.7	1.3	2.0
EM64T	2.7	1.4	2.0	2.7	0.7
Pentium4	2.3	1.8	2.7	0.5	2.2
Pentium M	2.0	0.8	1.9	1.0	2.6
G5	1.8	1.5	2.7	2.2	2.7
G5-Xserve	2.4	1.7	2.2	2.4	3.2

- **Typical accuracy within 4% in total time**
  - **some of the jobs run for over 1 hour**
- **In short**
  - **we do surprisingly well when averaging over the intricate details of cache and memory operations**

Use of the LPM: Architectural Studies

- Previous counts generated via hardware performance counters
  - We can also obtain counts using Cachegrind
- Cachegrind uses dynamic binary translation to capture all memory references and map them onto a user defined cache model
- Combine counts from Cachegrind with fitting parameters to predict performance as a function of cache architecture
  - Cache size
  - Line size
  - Associativity

Effect of Cache Size and Line Size: 2.2GHz AMD Opteron System

	Hardware	Cachegrind			
L1 Size	64KiB	64KiB	64KiB	64KiB	64KiB
L1 Line Size	64B	64B	32B	64B	64B
L2 Size	1MB	1MB	1MB	1MB	16MB
L2 Line size	64B	64B	64B	1024B	64B
Blocking Size	64KiW	64KiW	64KiW	64KiW	1MW
Icount/1E10	3.28	3.27	3.27	3.27	2.93
L2$Miss/1E6	9.77	7.03	7.03	3.2	0.04
Est Cycles/1E10	2.72	2.5	2.5	2.36	2.01

ANU

Current Work

- Exploring the domain of applicability
  - Fit for one system, but use on a very different
  - Eg fit for HF with 6-31g basis but use for an MP2 calculation with a 6-31++G(3df,3pd) basis
- Sensitivity of blocking factor to integral batch characteristics
  - Should we use the same blocking factor for [ss|ss] integrals as for a [dd|dd]
- Models to account for different latencies on NUMA architectures
  - On NUMA systems not all cache misses are equal
  - We have done many experiments using specific memory and thread placements to quantify these issues
  - Extending Cachegrind to include a memory placement model

ANU

ADAPTIVE
Dynamic Modification to Running Program

Objective: To develop a software environment that is responsive to changes in runtime conditions, simulation state, or information gathered by from other processes in the same system

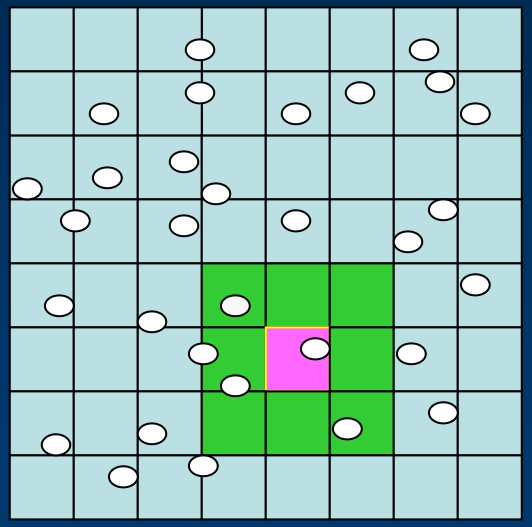
Example: Find all particles within a distance

- **Method 1:**

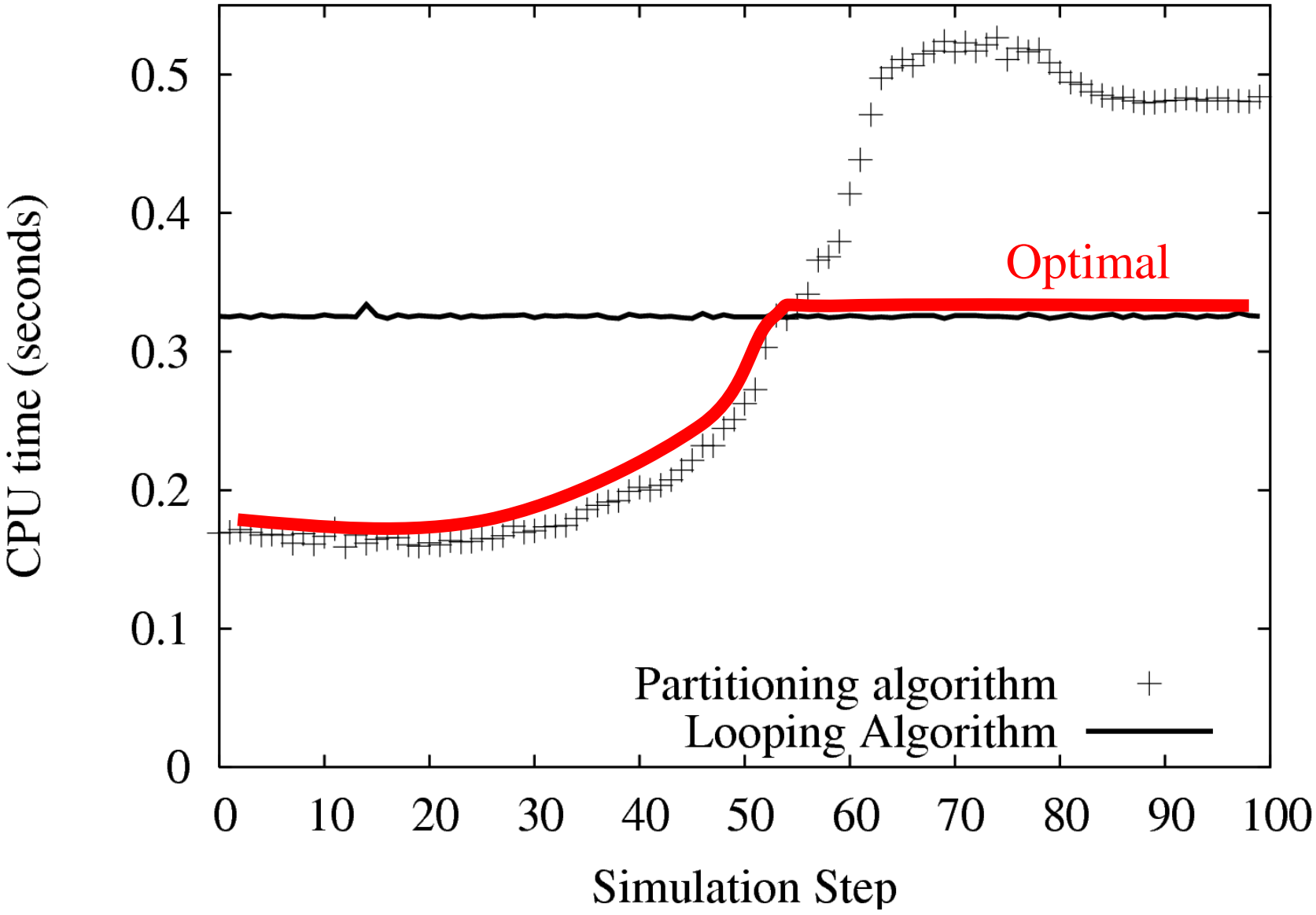
```
for i=2, N
   for j=1, j<I
      compute r_ij
      if (r_ij <
   end
end
```

  - Simple, but $O(N^2)$
- **Method 2:**
  - bound
  - p boxes
  - der adjacent boxes
  - More complex, but potentially $O(N)$

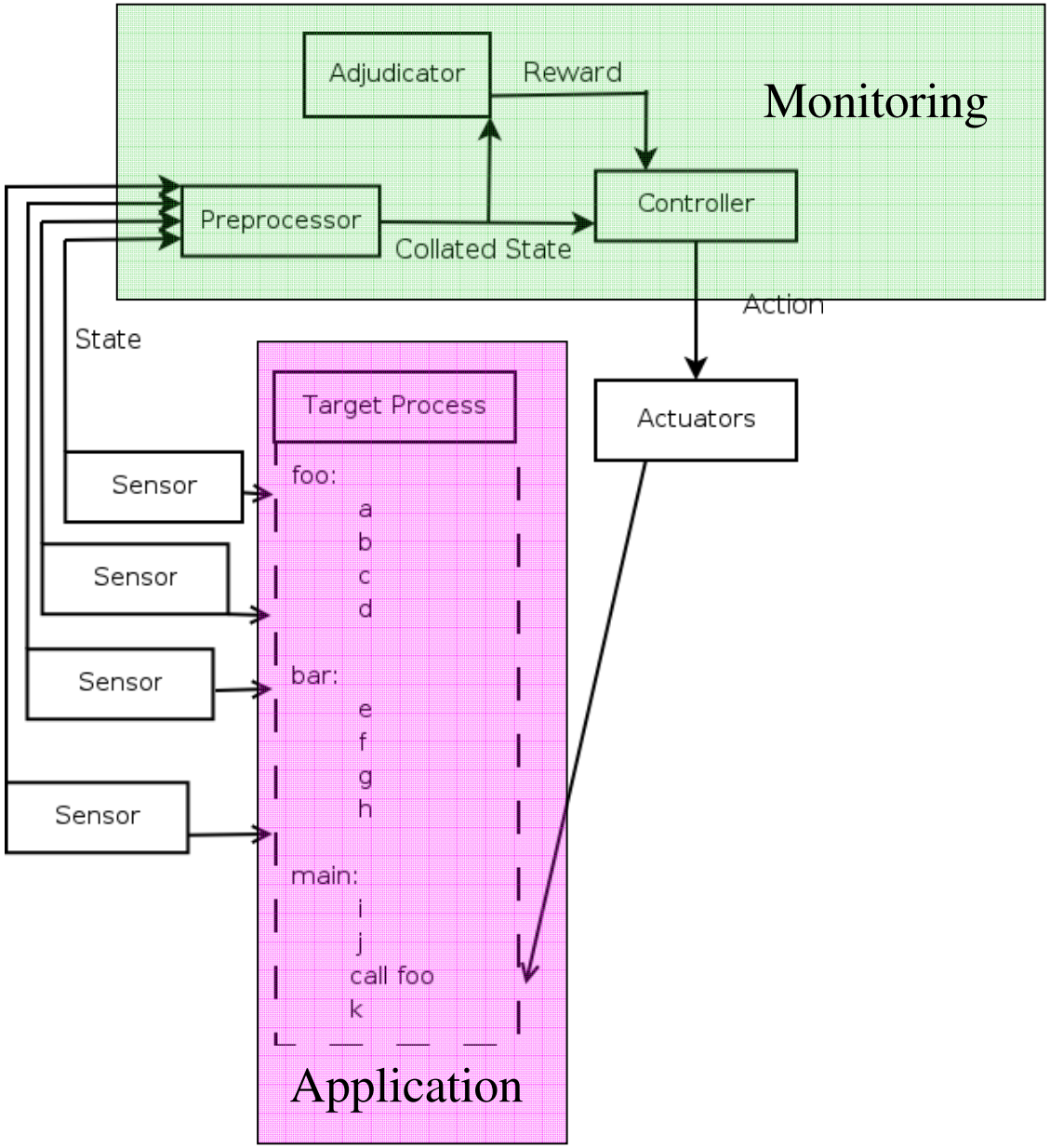
If particles are moving which method is faster can change with time



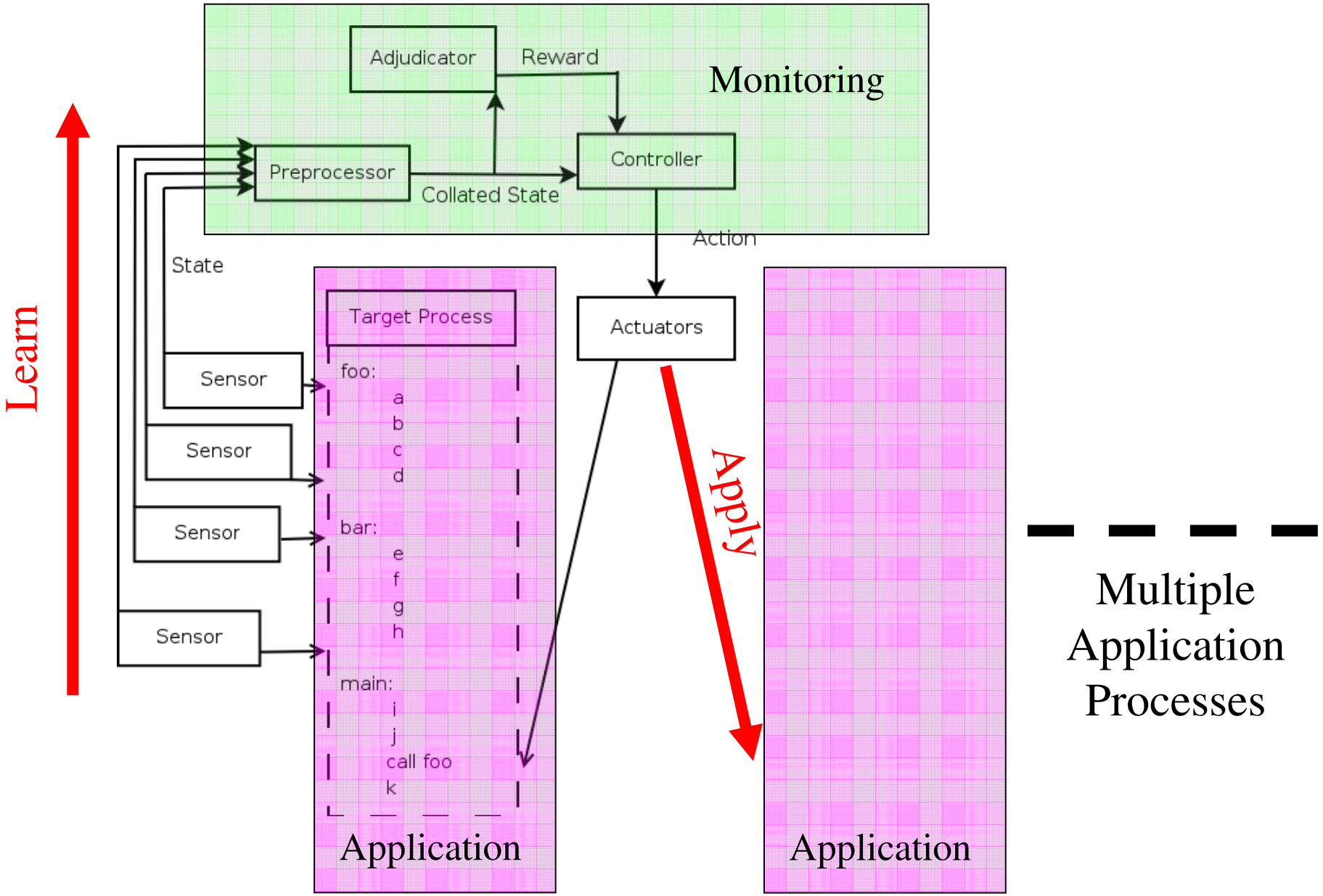
An artificial “Big Bang”



Model Architecture



Model Architecture

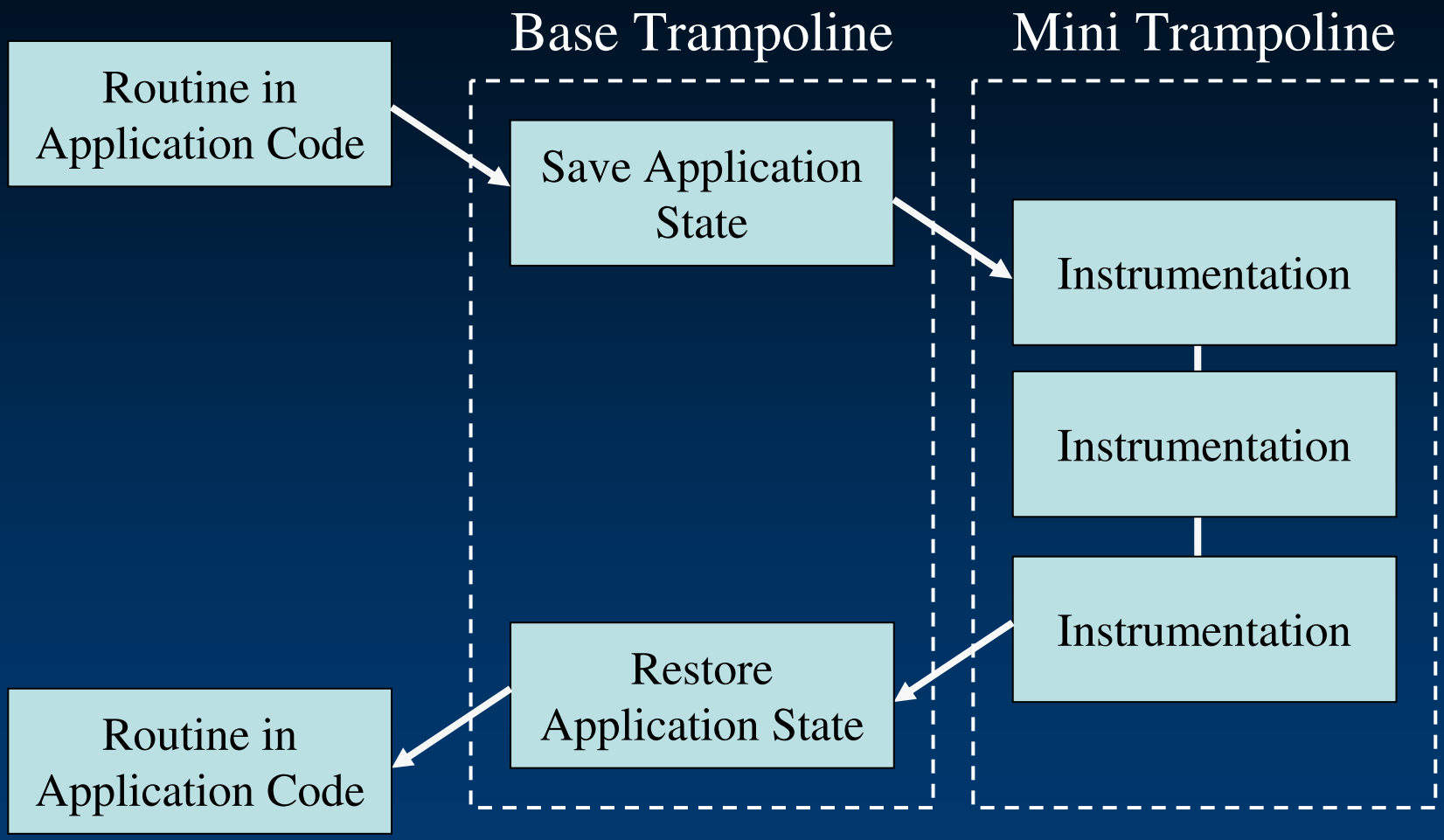


ANU

Implementational Requirements

- Need to be able to insert sensors into application and actuators to cause change
- Two possibilities:
  - Dynamic code modification
  - Use of a virtual machine (like JVM)
- We have been investigating use of DynInst (5.1)
- How do we decide on change?
  - Preliminary work based on use of reinforcement learning as this does not require a training set

Dynamic Code Modification with DynInst



Overhead of using DynInst on x86 System

	Safety Checks			
	None	Recursion	Floating Point	Rec+FP
Inline/loop header				
Wall time (ns)	37	39	159	162
CPU cycles	108	115	476	498
Instructions	12	19	20	27
Inline/called function				
Wall time (ns)	35	38	35	37
CPU cycles	102	109	102	109
Instructions	11	18	11	18
Outline/loop header				
Wall time (ns)	37	39	160	168
CPU cycles	110	116	476	501
Instructions	15	24	23	32
Outline/called function				
Wall time (ns)	35	38	35	38
CPU cycles	104	112	104	112
Instructions	14	23	14	23

Current Work

- Rigorous benchmarking for other parts of DynInst
  - Stopping/starting the code, copying data
- Investigating alternatives
  - Consideration of LLVM
- Initial target
  - Can we use reinforcement learning techniques to predict the optimal format for a sparse matrix based on a few key characteristics, and have the learner gather information from multiple running processes
  - Particularly interested in the sorts of sparse matrices that appear in large electronic structure calculations

Concluding Remarks

- After several years of relatively small changes in CPU architectures we are now seeing a flurry of new activity
  - Massively multicore
  - Heterogeneous
  - Complex memory hierarchies
- There is a need to re-think how we develop our software for these emerging systems
  - Models to predict performance
  - Quantifiable numerical errors
  - Strategies to automate the code tuning and optimization process
- We have put forward some ideas, but it is early days and there is much work left to do

Acknowledgements

- Australian Research Council Grants DP0558228, LP0669726 and LP0774896
- Sun Microsystems for equipment and discussions
- *Australian Partnership for Advanced Computing* and Alexander Technology for access to computing resources